Programming Assignment #5 – ARM Simulator
CSE30 - Computer Organization and Systems Programming
Winter 2011

Overview

This project involves creating an ARM simulator for a subset of the ARM instructions, which we call ARM LITE. The simulator is written in C. We have given you skeleton code (http://www.cse.ucsd.edu/~kastner/cse30/pa5.zip) that performs input and output and allocates the processor state (memory and registers). Your job is to write the `Simulator` function that executes code preloaded into your simulated memory space and performs the actions indicated by that code.

Project Description

You only need to handle a subset of the ARM instruction set. Any instructions outside of those listed below will not be used to test your program.

ARM LITE Instruction Set:

- Load/store instructions: `LDR, STR`
- Data Processing Instructions: `ADD, SUB, AND, ORR, EOR, CMP, MOV`
- Branch Instructions: `B, BX, BL`

Conditional Execution: You must be able to handle all 15 conditional execution of any instruction (`EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, AL`). This includes `ADDEQ, BNE, SUBGT`, etc.

Shifts and Rotates: You must be able to handle shifts and rotates (`LSL, LSR, ASR, ROR`) with constants (e.g., `ADD r0, r1, r3, LSL #4`) and with registers (e.g., `SUB r0, r1, r2, ROR r3`).

You must be able to handle the following addressing modes for LDR and STR instructions:

Addressing Mode	Example
Base Register	`LDR r0, [r1]`
Immediate Offset	`LDR r0, [r1, #4]`
Immediate Pre-Indexed	`LDR r0, [r1, #4]!`
Register Offset	`LDR r0, [r1, r2]`
Register Pre-Indexed	`LDR r0, [r1, r2]!`
Immediate Post-Indexed	`LDR r0, [r1], #4`
Register Post-Indexed	`LDR r0, [r1], r2`

Programming Assignment

You are given skeleton C code (PA5.c) that takes an input file, which is the code that you should execute in binary, and prints out the values in the registers and some simulator info into an output file. You are responsible for reading the instructions from memory, decoding them, executing them, writing back the results and updating the state of the processor. Your function should return when you find a `nop` instruction

You can compile the code using gcc. The corresponding program takes two arguments. The first is the input file (binary code) and the second argument is the output file (results of the execution).

All of your code goes into the `Simulator` function. You should not write code in any other part of the file. The lone argument to the `Simulator` function is an `ARMStruct * mem`.

`ARMStruct` has four fields:
`int registers[16];` Your simulated register files r0-r15
`int CPSR;` The current program status register
`char main_memory[4096];` Your memory. Note that your memory is 4 Kbytes
`int SIM_info[32];` A place to put some results of the simulation. This will be described later.

This `ARMStruct` has all of the processor state that we will look at in order to verify that you correctly designed your simulator. You must appropriately update all of this state in order to get full credit on the assignment. Any other processor state that you wish to keep is not important to us, but you are obviously free to keep track of it if you wish.

You must also collect some simulation data about the programs and store this in the `SIM_info` array. This includes:

`SIM_info[0]`	Number of times any LDR instruction is executed
`SIM_info[1]`	Number of times any STR instruction is executed
`SIM_info[2]`	Number of times any ADD instruction is executed
`SIM_info[3]`	Number of times any SUB instruction is executed
`SIM_info[4]`	Number of times any AND instruction is executed
`SIM_info[5]`	Number of times any ORR instruction is executed
`SIM_info[6]`	Number of times any EOR instruction is executed
`SIM_info[7]`	Number of times any CMP instruction is executed
`SIM_info[8]`	Number of times any MOV instruction is executed
`SIM_info[9]`	Number of times any B instruction is executed (not including BL or BX)
`SIM_info[10]`	Number of times any BL instruction is executed (not including B or BX)
`SIM_info[11]`	Number of times any BX instruction is executed (not including B and BL)
`SIM_info[12]`	Number of times any instruction with condition EQ was executed
`SIM_info[13]`	Number of times any instruction with condition NE was executed
`SIM_info[14]`	Number of times any instruction with condition CS was executed
`SIM_info[15]`	Number of times any instruction with condition CC was executed
`SIM_info[16]`	Number of times any instruction with condition MI was executed
`SIM_info[17]`	Number of times any instruction with condition PL was executed

SIM_info[18]	Number of times any instruction with condition VS was executed
SIM_info[19]	Number of times any instruction with condition HI was executed
SIM_info[20]	Number of times any instruction with condition LS was executed
SIM_info[21]	Number of times any instruction with condition GE was executed
SIM_info[22]	Number of times any instruction with condition LT was executed
SIM_info[23]	Number of times any instruction with condition GT was executed
SIM_info[24]	Number of times any instruction with condition LE was executed
SIM_info[25]	Number of times any instruction with condition AL was executed
SIM_info[26]	Number of times a Base Register memory instruction was executed
SIM_info[27]	Number of times a Immediate Offset memory instruction was executed
SIM_info[28]	Number of times a Immediate Pre-Indexed memory instruction was executed
SIM_info[29]	Number of times a Register Offset memory instruction was executed
SIM_info[30]	Number of times a Register Pre-Indexed memory instruction was executed
SIM_info[31]	Number of times a Immediate or Register Post-Indexed memory instruction was executed

For the conditional instructions, you should count an instruction regardless of whether or not the condition is true (i.e., it counts whether or not the result register is written).

Your code starts at memory address 0. You are expected to set your sp and pc register.

Generating Testcases

You are given one testcase in the pa5.zip folder. `test.bin` is the binary code that you should give as argument to your program. `test.s` shows the assembly corresponding to this code.

To compile your own test cases:

- download: http://www.modularcircuits.com/download/gcc-arm-elf-4.0.zip
- unpack
- using the `/bin` directory run commands below

assemble it:

```
arm-elf-as -o hello.o hello.s
```

link it:

```
arm-elf-ld -o hello.elf hello.o
```

save it out as a binary file:

```
arm-elf-objcopy -O binary hello.elf hello.bin
```

To run program

```
a.out in_file.bin out_file.txt
```

Hints:

- Make sure to setup your system initially in the same way as a real machine would be initialized.
- The ARM Instruction Document (http://cseweb.ucsd.edu/~kastner/cse30/arm-instructionset.pdf) provides a good overview of the instruction layout.

- Work on one instruction at a time. It is probably best to start with ADD since it is relatively straightforward.
- The `nop (0xE1A00000)` instruction indicates the end of your code.
- Your memory is a byte (char) array. Your instructions are 4 bytes (int).
- Test your code often.
- Start early.

Turnin Procedure

You are required to submit the entire `PA5.c` file. This includes the code given to you as well as your code in the `Simulator` function. The grading of the project will be done by changing the input program and observing the outputs. Therefore, you must store your results in the various `ARMStruct` fields, which will be printed out by the skeleton file. Failure to do this will result in loss of points.

When turning in your code, MAKE SURE YOU FOLLOW THIS FORMAT AND RENAME YOUR FILE:
a<student_id>.c
Failure to follow this format will result in loss of points.